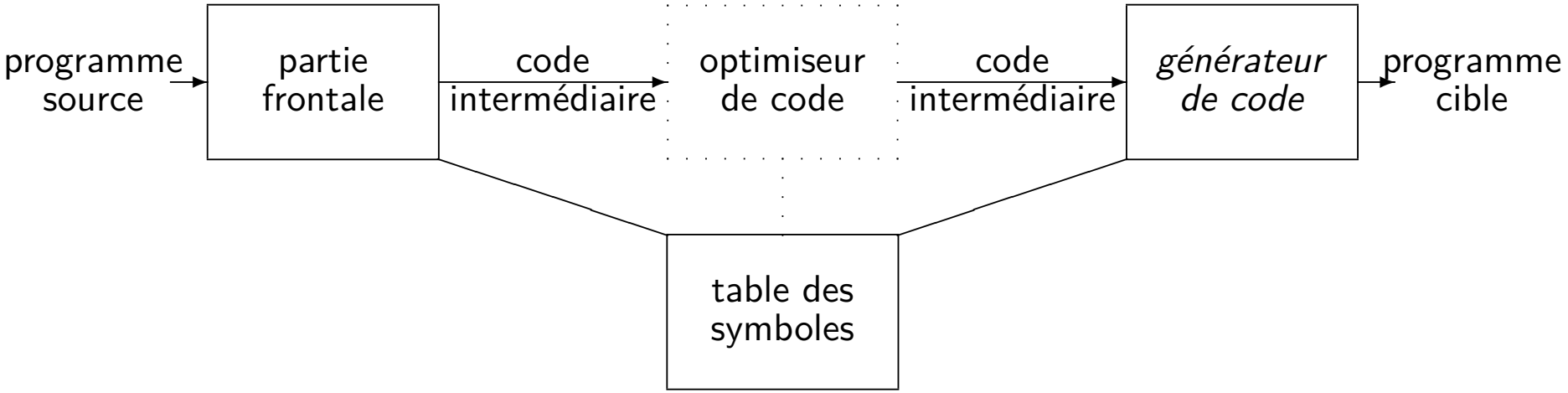
# Génération de code

Sections 8.1 à 8.5

# * Contenu *

- Objectifs du générateur de code
- Design du générateur de code
  - Langage cible
  - Gestion de la mémoire
  - Sélection des instructions
  - Allocation des registres
  - Choix de l'ordre d'évaluation
- Description de la machine cible utilisée dans le livre
- Gestion du rangement
  - Allocation statique
  - Allocation par pile
- Blocs de base et graphe de flot de contrôle

# Introduction




Le générateur de code:

- doit générer du code correct;
- devrait produire du code de bonne qualité;
- devrait être efficace;
- ne peut pas espérer générer un code optimal.

# Considérations liées au design du générateur de code

## Section 8.1

L'entrée fournie au générateur de code:

- Représentation intermédiaire du programme source et informations stockées dans la table des symboles.
- La représentation peut prendre la forme de:
  - suite linéaire d'opérations et d'opérandes en notation postfixe;
  - instructions à trois adresses;
  - instructions d'une machine à pile;
  - arbres (ou DAG) de syntaxe.
- Devrait avoir été analysée lexicalement, syntaxiquement et sémantiquement.
- Devrait être dotée d'opérations de conversion de types et de détection d'erreurs dynamiques.

# Considérations liées au design du générateur de code

La forme du programme cible est normalement une des suivantes:

- Langage machine à adressage absolu;
- Langage machine relocalisable;
- Langage assembleur;
- Langage pour une machine virtuelle
  - Exemples : JVM (Java virtual machine), .NET/CLI, LLVM
  - Généralement de plus haut niveau que le langage machine natif (fait abstraction du choix des registres, etc.)
- Autre langage de haut niveau.

Le fait de produire du code relocalisable ou assembleur permet d'effectuer de la *compilation par modules*.

# Considérations liées au design du générateur de code

Gestion de la mémoire pour les structures de données associées:

- à la table des symboles;
- aux étiquettes dans le code;
- aux noms des variables temporaires;
- aux informations produites par d'autres analyses de programmes;
- etc.

# Considérations liées au design du générateur de code

**Sélection des instructions.**

Il est aisé de faire une sélection d'instructions simple. On choisit un patron d'instructions adéquat pour chaque type de noeud de l'arbre de syntaxe abstraite.

Ex: pour l'énoncé `x := y + z` on peut choisir le patron

```
MOV R0,y        // R0 = y
ADD R0,R0,z     // R0 += z
MOV x,R0        // x = R0
```

On obtient alors du code de piètre qualité. Ex: pour

```
a := b + c
d := a + e
```

on obtient

```
MOV R0,b
ADD R0,R0,c
MOV a,R0
MOV R0,a
ADD R0,R0,e
MOV d,R0
```

C'est beaucoup plus complexe de faire un bon travail.

# Considérations liées au design du générateur de code

**Allocation des registres**.

On divise souvent ce problème en deux sous-problèmes:

- L'allocation des registres. *Quelles variables iront dans des registres?*
- L'affectation des registres. *Dans quel registre chaque variable va-t-elle?*

**Exemple:** On veut faire l'allocation des registres sur ce code intermédiaire.
(Supposons que les variables du programme source doivent toujours aller en mémoire.)

**Code interm.**

1. $t_1$ := b * b
2. $t_2$ := 4 * a
3. $t_3$ := $t_2$ * c
4. $t_4$ := $t_1$ - $t_3$
5. $t_5$ := sqrt $t_4$
6. $t_6$ := $t_5$ - b
7. $t_7$ := 2 * a
8. rt := $t_6$ / $t_7$

# Considérations liées au design du générateur de code

**Allocation des registres. Exemple:**

**Code interm.**

1. $t_1$ := b * b
2. $t_2$ := 4 * a
3. $t_3$ := $t_2$ * c
4. $t_4$ := $t_1$ - $t_3$
5. $t_5$ := sqrt $t_4$
6. $t_6$ := $t_5$ - b
7. $t_7$ := 2 * a
8. rt := $t_6$ / $t_7$

| Variable | Alloc. |
|---|---|
| $t_1$ | reg. |
| $t_2$ | reg. |
| $t_3$ | reg. |
| $t_4$ | reg. |
| $t_5$ | reg. |
| $t_6$ | reg. |
| $t_7$ | reg. |
| a | mem. |
| b | mem. |
| c | mem. |
| rt | mem. |

En fait, décréter

qu'on veut qu'une variable soit allouée dans les registres

signifie en réalité

qu'on veut qu'une variable soit allouée dans les registres *si possible*.

C'est que la quantité de registres n'est pas illimitée.

# Considérations liées au design du générateur de code

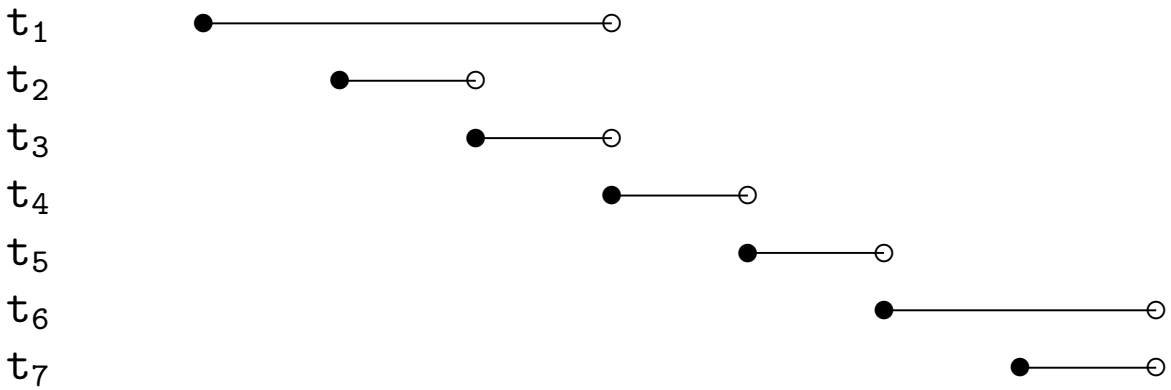
## Allocation des registres. Exemple:

**Code interm.**

1. $t_1$ := b * b
2. $t_2$ := 4 * a
3. $t_3$ := $t_2$ * c
4. $t_4$ := $t_1$ - $t_3$
5. $t_5$ := sqrt $t_4$
6. $t_6$ := $t_5$ - b
7. $t_7$ := 2 * a
8. rt := $t_6$ / $t_7$

| Variable | Alloc. | Durée vie |
|---|---|---|
| $t_1$ | reg. | 1–4 |
| $t_2$ | reg. | 2–3 |
| $t_3$ | reg. | 3–4 |
| $t_4$ | reg. | 4–5 |
| $t_5$ | reg. | 5–6 |
| $t_6$ | reg. | 6–8 |
| $t_7$ | reg. | 7–8 |
| a | mem. | $\infty$ |
| b | mem. | $\infty$ |
| c | mem. | $\infty$ |
| rt | mem. | $\infty$ |

Les durées de vie, graphiquement:



# Considérations liées au design du générateur de code

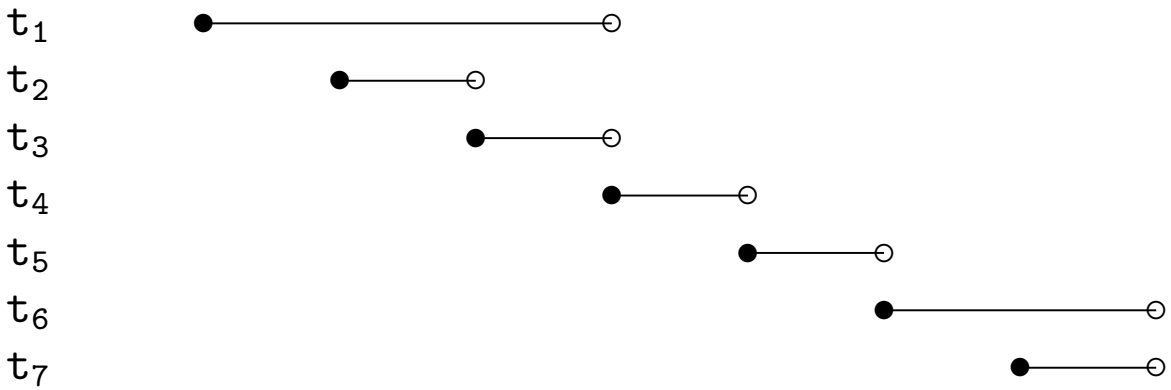
**Allocation des registres. Exemple:**

**Code interm.**

1. $t_1$ := b * b
2. $t_2$ := 4 * a
3. $t_3$ := $t_2$ * c
4. $t_4$ := $t_1$ - $t_3$
5. $t_5$ := sqrt $t_4$
6. $t_6$ := $t_5$ - b
7. $t_7$ := 2 * a
8. rt := $t_6$ / $t_7$

| Variable | Alloc. | Durée vie | Affect. |
|---|---|---|---|
| $t_1$ | reg. | 1–4 | R1 |
| $t_2$ | reg. | 2–3 | R2 |
| $t_3$ | reg. | 3–4 | R2 |
| $t_4$ | reg. | 4–5 | R1 |
| $t_5$ | reg. | 5–6 | R1 |
| $t_6$ | reg. | 6–8 | R1 |
| $t_7$ | reg. | 7–8 | R2 |
| a | mem. | $\infty$ | — |
| b | mem. | $\infty$ | — |
| c | mem. | $\infty$ | — |
| rt | mem. | $\infty$ | — |

Les durées de vie, graphiquement:



# Considérations liées au design du générateur de code

**Allocation des registres**.

L'affectation optimale des registres est un problème NP-complet.

En pratique, on se contente d'heuristiques rapides et d'assez bonne qualité.

Des machines aux instructions irrégulières compliquent la tâche.

- Les instructions irrégulières sont des instructions dont la source et/ou la destination ne sont pas (ou pas complètement) configurables.
- Exemple: l'instruction `div` de la famille `x86` d'Intel reçoit toujours le dividende dans les registres `AX` et `DX`, et place toujours le quotient dans le registre `AX`.

Algorithm:

when operand is a **byte**:
AL = AX / operand
AH = remainder
(modulus)

when operand is a **word**:
AX = (DX AX) / operand
DX = remainder
(modulus)

**Source:** `http://www.electronics.dit.ie/staff/tscarff/8086_instruction_set/8086_instruction_set.html`

# Considérations liées au design du générateur de code

## Choix de l'ordre d'évaluation

- Certaines constructions des langages sources laissent l'ordre d'évaluation indéterminé.
- Or, certains ordres d'évaluation peuvent s'avérer plus efficaces que d'autres.

**Exemple 1**. Soit 'a' une variable de type entier positif.

```
// Ordre original
if (sqrt(a) > 200 && even(a))
  { f(); }
```

```
// Ordre plus avantageux
if (even(a) && sqrt(a) > 200)
  { f(); }
```

La fonction `pair` s'exécute beaucoup plus rapidement que la fonction `sqrt`, on pourrait donc vouloir exécuter `pair` en premier; ainsi, si le nombre est impair, on peut éviter l'appel à `sqrt` (cette optimisation est possible car les fonctions `pair` et `sqrt` sont définies pour tout entier positif et n'ont pas d'effet de bord).

# Considérations liées au design du générateur de code

## Choix de l'ordre d'évaluation

**Exemple 2**. Supposons que nous générons du code pour un processeur x86 (intel) et que nous utilisons l'instruction `div` pour effectuer les divisions. (Rappelons-nous que l'instruction 'inc' fonctionne plus rapidement sur un registre que sur une adresse mémoire).

```
// (a) Ordre
// original
a := a / x
b := b / y
a := a + 1
b := b + 1

// (b) Ordre
// plus avantageux
a := a / x
a := a + 1
b := b / y
b := b + 1
```

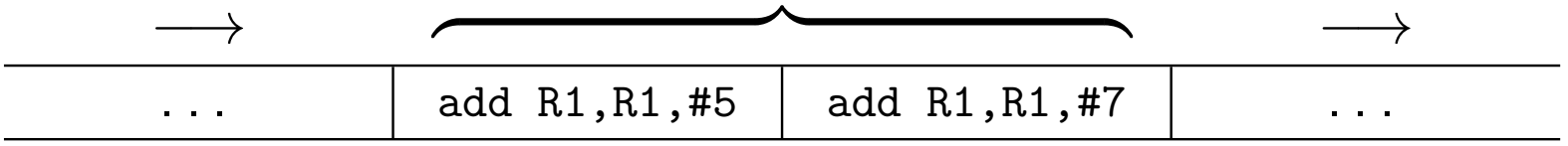
```
// (a) En
// assembleur
mov eax, a
mov bx,  x
div bx
mov a,   ax
mov eax, b
mov bx,  y
div bx
mov b,   ax
inc a
inc b
```

```
// (b) En
// assembleur
mov eax, a
mov bx,  x
div bx
inc ax
mov a,   ax
mov eax, b
mov bx,  y
div bx
inc ax
mov b,   ax
```

# Considérations liées au design du générateur de code

Objectifs/approches de la génération de code:

- Correction.
- Maximisation de l'utilisation des registres.
- Optimisation *peephole*. Exemple:




  Le code dans la fenêtre est remplacé par: `add R1,R1,#12`.

- Analyse de flot de contrôle.
- Pour la vitesse et/ou l'efficacité.
- Une myriade d'autres...

Génération de code:

# Description de la machine cible utilisée dans le livre

# Machine cible

## Section 8.2

Machine à $n$ registres avec des instructions "à trois adresses" de la forme générale:

$$op \quad destination, source_1, source_2$$

où *op* est quelque chose comme `MOV`, `ADD` ou `SUB`.

Divers modes d'adressage sont fournis:

| Mode | Forme | Adresse | Coût additionnel |
|---|---|---|---|
| variable | $x$ | $\&x$ | 1 |
| indexé par variable | $x$(R) | $\&x + contenu$(R) | 1 |
| indexé par décalage | $c$(R) | $c + contenu$(R) | 1 |
| indirect | *R | $contenu$(R) | 0 |
| immédiat | #$c$ | $c$ | 1 |

# Coût des instructions

Les instructions ont naturellement un coût relié à leur exécution. Par exemple, une division réelle vs. une addition entière. Par exemple, une instruction avec accès en mémoire vs. sans accès.

Mais aussi, il y a un coût relié au *chargement* de l'instruction à partir de la mémoire.

Par exemple, une instruction qui inclut une adresse absolue est plus longue et est donc plus coûteuse à charger.

Même chose pour une instruction qui inclut une constante litérale.

Génération de code:

# Gestion du rangement dans le code cible

# Gestion du rangement dans le code cible

## Section 8.3

Cette section se concentre sur le code qu'il faut générer pour implanter la mécanique d'appel.

On suppose qu'on a les instructions à trois adresses suivantes:

1. `call`;
2. `return`;
3. `halt` et
4. `action` qui symbolise toute instruction qui n'est pas reliée à la mécanique d'appel.

# Gestion du rangement dans le code cible

## Allocation statique

L'instruction intermédiaire `call` est transformée en les instructions machines suivantes:

```
ST   callee.staticArea, #here + 20      // MOV
BR   callee.codeArea                    // GOTO
```

Ces instructions effectuent les actions suivantes:

- Placer l'adresse de retour dans le bloc d'activation de la fonction appelée.$^{(*)}$
- Passer le contrôle à l'appelé.

$(*)$ On suppose que le code à exécuter au retour se trouve juste après le `BR` et que la taille des instructions `ST` et `BR` est de 20 octets.

L'instruction intermédiaire `return` est transformée en l'instruction machine:

```
BR   *callee.staticArea
```

Celle-ci récupère, dans le bloc d'activation de la fonction appelée, l'adresse de retour chez l'appelant.

# Allocation statique : Exemple 8.3

```
// Code de 'c' (supposons qu'il commence à l'adresse 100
//   et que sa structure d'activation est à l'adresse 300)
action 1  // 20 octets d'instructions quelconques
call p
action 2  // 20 octets d'instructions quelconques
halt

// Code de 'p' (supposons qu'il commence à l'adresse 200
//   et que sa structure d'activation est à l'adresse 364)
action 3  // 20 octets d'instructions quelconques
return
```

**Code généré:**

```
100: ACTION 1
120: ST 364, #140  // Sauver l'adresse de retour (140) à 364
132: BR 200        // invoquer p
140: ACTION 2
160: HALT
...
200: ACTION 3
220: BR *364       // Retourner à l'adresse sauvée à 364
```

# Gestion du rangement dans le code cible

## Allocation par pile

L'instruction intermédiaire `call` est transformée en les instructions machines suivantes:

```
ADD  SP, SP, #caller.recordSize     // SP += #caller.recordSize
ST   *SP, #here + 16                // *SP = #here + 16
BR   callee.codeArea                // GOTO callee.codeArea
SUB  SP, SP, #caller.recordSize     // SP -= #caller.recordSize
```

Ces instructions effectuent les actions suivantes:

- Faire sauter le pointeur de pile `SP` par-dessus le bloc d'activation de l'appelant. (**Note:** ici, on suppose que `SP` pointe toujours *au début* du bloc de la fonction en cours d'exécution.)
- Placer l'adresse de retour dans le bloc d'activation de la fonction appelée (à la première case du bloc).
- Passer le contrôle à l'appelé.
- [La dernière instruction fait partie du protocole de retour.]

# Gestion du rangement dans le code cible

## Allocation par pile

L'instruction intermédiaire `return` est transformée en l'instruction machine:

```
BR  *0(SP)    // GOTO *SP
```

Cette instruction ainsi que la quatrième chez l'appelant effectuent les actions suivantes:

- Récupérer l'adresse de retour dans le bloc d'activation courant (celui de l'appelé) et passer le contrôle à l'appelant à l'endroit spécifié.
- Dépiler le bloc d'activation de l'appelé en ramenant `SP` au début du bloc de l'appelant.

# Gestion du rangement dans le code cible

## Allocation par pile

Le code généré pour le programme principal a la forme suivante:

```
LD     SP, #stackStart          // MOV
code du programme principal
HALT
```

# Allocation par pile : Exemple 8.4

**Source**

```
// Code de ‘m’
// (bloc de taille ‘ms’)
action 1
call q
action 2
halt

// Code de ‘p’
// (bloc de taille ‘ps’)
action 3
return

// Code de ‘q’
// (bloc de taille ‘qs’)
action 4
call p
action 5
call q
action 6
call q
return
```

**Code généré** (Supposons que la pile commence à 600)

```
// Code de ‘m’
   100: LD SP, #600      // initialiser la pile
   108: ACTION 1
   128: ADD SP, SP, #ms  // SP += ms
   136: ST *SP, #152     // *SP = (adr. retour)
   144: BR 300           // appeler q
   152: SUB SP, SP, #ms  // SP -= ms
   160: ACTION 2
   180: HALT
// Code de ‘p’
   200: ACTION 3
   220: BR *0(SP)        // return
// Code de ‘q’
   300: ACTION 4
   320: ADD SP, SP, #qs  // SP += qs
   328: ST *SP, #344     // *SP = (adr. retour)
   336: BR 200           // appeler p
   344: SUB SP, SP, #qs  // SP -= qs
   352: ACTION 5
   ...
   456: BR *0(SP)        // return
```

Génération de code:

# Blocs de base et graphe de flot de contrôle

# Blocs de base

## Section 8.4

Un *bloc de base* est une suite d'instructions consécutives dans laquelle le contrôle ne peut qu'entrer par la première, où l'exécution ne peut s'arrêter et où un branchement ne peut se produire qu'à la dernière instruction.

**Exemple:** cette séquence d'instructions est un bloc de base si on suppose qu'aucune instruction de branchement ne peut faire passer le contrôle directement à une des instructions (exceptée peut-être la première):

$$
\begin{aligned}
\mathtt{t}_1 &:= \mathtt{a} * \mathtt{a} \\
\mathtt{t}_2 &:= \mathtt{a} * \mathtt{b} \\
\mathtt{t}_3 &:= 2 * \mathtt{t}_2 \\
\mathtt{t}_4 &:= \mathtt{t}_1 + \mathtt{t}_3 \\
\mathtt{t}_5 &:= \mathtt{b} * \mathtt{b} \\
\mathtt{t}_6 &:= \mathtt{t}_4 + \mathtt{t}_5
\end{aligned}
$$

On dit que l'instruction `x := y + z` *définit* `x` et *utilise* (ou *référence*) `y` et `z`.

On dit qu'une variable est *vivante* en un certain point si cette variable est utilisée à nouveau ultérieurement, possiblement dans un autre bloc de base.

# Blocs de base

## Algorithme 8.5

*Entrée.* Une suite d'instructions à trois adresses.

*Sortie.* Une liste de blocs de base où chaque instruction apparaît une fois dans exactement un bloc.

*Méthode.*

1. On identifie tout d'abord les *instructions de tête* en se fiant aux règles suivantes:
   (a) La première instruction est une instruction de tête.
   (b) Toute instruction qui est la destination d'une instruction de branchement, conditionnel ou inconditionnel, est une instruction de tête.
   (c) Toute instruction qui suit immédiatement une instruction de branchement est une instruction de tête.
2. Pour chaque instruction de tête, extraire le bloc de base qui va de l'instruction de tête, inclusivement, jusqu'à la prochaine instruction de tête, exclusivement, ou jusqu'à la fin du programme.

# Blocs de base

**Exemple 8.6**. Ce morceau de code change une matrice 10x10 en matrice identité:

```
for i from 1 to 10 do
    for j from 1 to 10 do
        a[i,j] = 0.0;
for i from 1 to 10 do
    a[i,i] = 1.0;
```

Le code intermédiaire correspondant est:

```
1. i = 1
2. j = 1
3. t1 = 10 * i
4. t2 = t1 + j
5. t3 = 8 * t2
6. t4 = t3 - 88
7. a[t4] = 0.0
8. j = j + 1
9. if j <= 10 goto 3
10. i = i + 1
11. if i <= 10 goto 2
12. i = 1
13. t5 = i - 1
14. t6 = 88*t5
15. a[t6] = 1.0
16. i = i + 1
17. if i <= 10 goto 13
```

# Blocs de base

**Exemple 8.6**. Les instructions de tête sont :

- L'instruction 1, car elle est la première;
- Les instructions 2, 3 et 13, car elles sont la destination d'un saut.
- Les instructions 10 et 12, car elles suivent un saut (conditionnel);

```
1. i = 1
2. j = 1
3. t1 = 10 * i
4. t2 = t1 + j
5. t3 = 8 * t2
6. t4 = t3 - 88
7. a[t4] = 0.0
8. j = j + 1
9. if j <= 10 goto 3
```

```
10. i = i + 1
11. if i <= 10 goto 2
12. i = 1
13. t5 = i - 1
14. t6 = 88*t5
15. a[t6] = 1.0
16. i = i + 1
17. if i <= 10 goto 13
```

# Graphes de flot de contrôle

On peut compléter l'information de flot de contrôle manquant à un ensemble de blocs de base en construisant un *graphe de flot de contrôle*.

Chaque noeud du graphe est un bloc de base.

Un des noeuds est le noeud *initial*; c'est celui qui contient la première instruction.

Il y a un arc du bloc $B_1$ au bloc $B_2$ si:

- il y a une instruction de branchement de la dernière instruction de $B_1$ à la première instruction de $B_2$; ou
- $B_2$ suit immédiatement $B_1$ dans le programme et $B_1$ ne se termine pas sur une instruction de branchement inconditionnel.

On dit que $B_1$ est un *prédécesseur* de $B_2$ et que $B_2$ est un *successeur* de $B_1$.

**Boucle** : on dit qu'un sous-ensemble de blocs du graphe de flot de contrôle forme une *boucle* si ces blocs sont fortement connectés et s'il existe une seule *entrée* au sous-ensemble de blocs.

# Graphes de flot de contrôle

**Exemple 8.8**. Le code présenté à l'exemple 8.6 forme le graphe de flot de contrôle suivant :